GD2S03
Advanced Software Engineering & Programming for Games

Bachelor of Software Engineering(BSE)
Game Development

- Overview
  - Properties
  - Methods
  - Subscripts
  - Inheritance

Properties

- Values associated with a particular class, structure or enumerations. Can be of two types - stored and computed
- **Stored Property** – store constant or variable values as part of an instance.
- **Computed Property** – calculate a value
- Stored and computed properties are associated with an instance.
- **Type properties** are associated with the type itself.
- Change in property's value can be monitored by **property observers**.

2.2.1 Stored Properties

Stored properties can be either *variable stored properties* (introduced by the **var** keyword) or *constant stored properties* (introduced by the **let** keyword).

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int     //constant
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
```

Properties

Stored Properties of Constant Structure Instances

- Structures - *value types* - constant properties if let.
- Classes-*reference types*- can change variable properties.

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// error, even though firstValue is a variable property
```

Lazy Stored Properties

- A *lazy stored property* is a property whose initial value is not calculated until the first time it is used.
- Indicated by writing the *lazy* modifier before its declaration

```
class getNumber{
    var temp = 10;

}
```

```
class Manage{
    var number = [Int]();
    lazy var result = getNumber();
}
```

```
let manage = Manage()
manage.number.append(1) // getNumber instance has not yet called
manage.result.temp; // getNumber Called
```

Properties

Computed Properties

- In addition to stored properties, classes, structures, and enumerations can define *computed properties*.
- Computed Properties do not actually store a value.
- Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

```swift
struct FindCenter{
    var origin = (0,0), size = (10, 10)
    var center:(Int, Int){
        set(newCenter){
            origin.0 = newCenter.0 - (size.0/2)
            origin.1 = newCenter.1 - (size.1/2)
        }
        get{
            let centerX = origin.0 + (size.0/2)
            let centerY = origin.1 + (size.1/2)
            return (centerX, centerY)
        }
    }
}
var findCenter = FindCenter()
print("Default Center is \(findCenter.center)")
//Default Center is (5, 5)

findCenter.center = (15, 15)

print("origin is \(findCenter.origin.0, findCenter.origin.1)")
//origin is (10, 10)
```

Properties

Shorthand Setter Declaration

If a computed property's setter does not define a name for the new value to be set, a default name of ***newValue*** is used. Here's an alternative version of the **Rect** structure, which takes advantage of this shorthand notation:

```
var center:(Int, Int){
        set{
            origin.0 = newValue.0 - (size.0/2)
            origin.1 = newValue.1 - (size.1/2)
        }
        get{
            let centerX = origin.0 + (size.0/2)
            let centerY = origin.1 + (size.1/2)
            return (centerX, centerY)
        }
    }
```

Read Only Computed Properties

- A computed property with a getter but no setter.
- A read-only computed property always returns a value.
- Can be accessed through dot syntax, but cannot be set to a different value.
- Declared as `var` because their value is not fixed.

```
struct FindArea{
    var length = 10, width = 20
    var area:Int{
        return length * width
    }
}
var findArea = FindArea()
print("Area is \(findArea.area)")
```

Properties

Property Observers

- Property observers observe and respond to changes in a property's value
- ***willSet*** is called just before the value is stored.
- ***didSet*** is called immediately after the new value is stored.
- When a property is set in an initializer willset and didset observers cannot be called.
- Except lazy stored properties, property observers can be added to 'inherited' property by method 'overriding'.
- Local constants and variables are never computed lazily.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200


// About to set totalSteps to 200
// Added 200 steps
```

Properties

Type Properties

- Properties that belong to the type itself, not to any one instance of that type.
- Unlike stored instance properties, stored type properties must always be given a default value.
- Lazily initialized.
- Define type properties with the ***static*** keyword.

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

Properties

- For computed type properties for class types, use the **class** keyword instead to allow subclasses to override the superclass's implementation.

- The computed type property examples are for read-only computed type properties, but can also define read-write computed type properties with the same syntax as for computed instance properties.

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value.”
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27”


class AnotherClass: SomeClass{
    var name: String?
    override class var overrideableComputedTypeProperty: Int {
        return 507
    }

   //Cannot override static var
    override static var computedTypeProperty: Int{
        return storedTypeProperty * 10
    }
}
```

Methods

Instance Methods

- Functions that are instances of a particular class, structure, or enumeration.
- Providing ways to access and modify instances
- Dot syntax is used to call the instance method:

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
let counter = Counter()// the initial counter value is 0
counter.increment() // the counter's value is now 1
```

- The Self Property
- *self* property is used to refer to the current instance within its own instance methods.

```
class Counter {
    var count = 0

    func increment() {
        self.count += 1
    }
}
```

Methods – Modifying value types from within instance methods

- Structures and enumerations are *value types*. By default, the properties of a value type cannot be modified from within its instance methods.

- However, to modify the properties of structure or enumeration within a particular method, opt in to ***mutating*** behavior for that method

- Mutating methods can assign an entirely new instance to the implicit **self** property.
  The **Point** example could have been written in the following way instead:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double){
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)“
-------------------------------------------------------------
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)

print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)“
```

Methods – Mutating methods for enumerations

- Can set the implicit **self** parameter to be a different case from the same enumeration:

- This example defines an enumeration for a three-state switch. The switch cycles between three different power states (**off**, **low** and **high**) every time its **next()** method is called.

```swift
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next() // ovenLight is now equal to .high
ovenLight.next() // ovenLight is now equal to .off
```

Methods – Type Methods

- Methods that are called on type itself, Indicated by *static* keyword.
- Classes may also use the **class** keyword to allow subclasses to override the superclass's implementation of that method.
- Within the body of a type method, the implicit ***self*** property refers to the type itself, rather than an instance of that type.
- A type method can call another type method with the other method's name, without needing to prefix it with the type name.

```swift
class SomeClass {
    class func overridableType() {
        print("inside overridable")
    }

    static func nonOverridableType(){
        print("inside non overridable")
    }
}

print(SomeClass.overridableType())

class AnotherClass: SomeClass{
    override class func overridableType(){
        print("inside overridable subclass")
    }

   //Cannot override static method

    override static func nonOverridableType(){
        print("inside non overridable subclass")
    }
}
```

Methods - Example

```swift
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1
    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel {
            highestUnlockedLevel = level
        }
    }
    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }

    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        }
        else {
            return false
        }
    }
}
```

```swift
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now
\(LevelTracker.highestUnlockedLevel)")
// Prints "highest unlocked level is now 2“
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// Prints "level 6 has not yet been unlocked"
```

Subscripts

- Classes, structures, and enumerations can define *subscripts*.
- Subscripts are shortcuts for accessing the member elements of a collection, list, or sequence.
- Defined using **subscript** keyword, and like instance methods specify one or more input parameters and a return type.
- Syntax is similar to both instance method syntax and computed property syntax.
- Subscripts enables to query instances of a type by writing one or more values in square brackets after the instance name.
- Subscripts can be defined for multiple dimensions with multiple input parameters.
- Subscripts can be read-write or read-only. This behavior is communicated by a getter and setter.

```
subscript(index: Int) -> Int {
    get {
  // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}


struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
// Prints "six times three is 18"
```

Subscripts

- The exact meaning of “subscript” depends on the context in which it is used
- Typically used as a shortcut for accessing the member elements in a collection, list, or sequence.
- You are free to implement subscripts in the most appropriate way for your particular class or structure’s functionality.

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

*Swift’s **Dictionary** type implements its key-value subscripting as a subscript that takes and returns an optional type.*

Subscript Options

- Subscripts can take any number of input parameters, and these input parameters can be of any type.
- Subscripts can also return any type.
- Subscripts can use ***variadic*** parameters.
- Can’t use ***in-out*** parameters or provide default parameter values.
- A class or structure can provide as many subscript implementations as it needs
- This definition of multiple subscripts is known as *subscript overloading*.

Subscripts - Example

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
  return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column),
                                    "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column),
                                    "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

```
var matrix = Matrix(rows:
2, columns: 2)
matrix[0, 1] = 1.5 // set
the value
let value = matrix[1, 1] //
get the value
//let someValue = matrix[2,
2]
// this triggers an assert,
because [2, 2] is outside
of the matrix bounds
```

Inheritance

- A class can *inherit* methods, properties, and other characteristics from another class.
- Classes can also add property observers to inherited properties in order to be notified when the value of a property changes.
- Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing
    }
}


let someVehicle = Vehicle()
print("Vehicle: \(someVehicle.description)")
```

Inheritance - Subclassing

- *Subclassing* is the act of basing a new class on an existing class.

- The subclass inherits characteristics from the existing class.

- Also new characteristics can be added to the subclass.

- To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon:

- Subclasses can themselves be subclassed.

```
class Bicycle: Vehicle {
    var hasBasket = false
}


let bicycle = Bicycle()
bicycle.hasBasket = true
bicycle.currentSpeed = 15.0


print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour


class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)") // 22.0 muiles
```

Inheritance – Overriding

Accessing Superclass Methods, Properties, and Subscripts

- Where its' apprpriate, superclass version of a method, property or subscript can be accessed by ***super*** keyword.

- An overridden method named someMethod() can call the superclass version of someMethod() by calling super.someMethod() within the overriding method implementation.
- An overridden property called someProperty can access the superclass version of someProperty as super.someProperty within the overriding getter or setter implementation.
- An overridden subscript for someIndex can access the superclass version of the same subscript as super[someIndex] from within the overriding subscript implementation.

Overriding methods

- An inherited instance or type method can be overridden to provide a tailored or alternative implementation of the method within the subclass

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

Inheritance - Overriding

Overriding property getters and Setters

- Overridden property(stored and computed) can be provided with custom getter and setter
- However both the name and type of the overridden property should be provided.
- Read only property can be inherited as read-write by providing both getter and setter.
- However read-write property cannot be overridden as read-only.
- Setter property can not be provided alone for any overriding property(it comes with getter as well)

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
 }
class Car: Vehicle {
    var gear = 1
    override var currentSpeed: Double{
        get {  return super.currentSpeed   }
        set {  super.currentSpeed = newValue }
     }
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}

let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

Inheritance – Overriding

Overriding Property Observers

- Property overriding can be used to add property observers to an inherited property.
- Property observers cannot be added to inherited constant stored properties or inherited read-only computed properties.
- Both an overriding setter and an overriding property observer for the same property cannot be provided as any value change can be observed in custom setter.

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}


let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles
// per hour in gear 4
```

Preventing Overrides

- A method, property, or subscript can be prevented from being overridden by marking it as *final*.
- An entire class as can be marked final by writing the final modifier before the class keyword.